**CS 345 Programming Languages**

15: Overview of Functional Programming

---

**Pure vs. Impure**
- **Pure**: No side effects
  - Example: Haskell
- **Impure**: Has side effects
  - Example: ML
    - Print
    - Assignment
- Litmus test for side effect:
  \[ \forall f : f \Rightarrow f \circ f \]

---

**Mutability vs. Immutability of Data**
- **ML**:
  - `val x = ref 5; x := 10; val y = !x + 1;`
  - `val x = ref 5 : int ref`
  - `val it = () : unit`
  - `val y = 11 : int`
- **Prolog**:
  - `put_assoc(FnName,EnvIn,FnBody,EnvOut).`
- **Arrays are often mutable data structures**

---

**Lazy Evaluation**
- `quickSort [] = []`
- `quickSort (x:xs) = quickSort (filter (< x) xs) ++ [x] ++ quickSort (filter (>= x) xs)`
- `minimum ls = head (quickSort ls)`
- In practice purity is almost a prerequisite for lazy evaluation
- Many imperative languages can lazily evaluate conditions:
  - `if (a && b) ...`  
  - `if (a || b) ...`
Type System

- Some languages like part of the Lisp Family are based on an untyped $\lambda$-Calculus.
  - Also Scheme

- Some languages like the ML Family use a typed $\lambda$-Calculus and a Hindley-Milner Type System
  - Also Haskell, with some modifications

- Many languages provide algebraic data types (tuples, records, ...) as a first class citizen
  - Except for the Lisp Family (only Lists)

Explicit vs. Implicit Typing

- Explicit vs. Implicit
  - Explicit: Type annotations
    - Some languages only support explicit type declaration
      - Java, C, Ada, Pascal, C++, ...
  - Implicit: Type inference
    - ML, OCaml, Haskell, F#, Scala, Go, C++11 (auto), ...
    - Most languages with type inference also support explicit type annotations

Static vs. Dynamic Typing

- Static vs. Dynamic
  - Static: Compile time
    - Java, Scala, Haskell, ML, C, C++
  - Dynamic: Runtime
    - Ruby, Lisp Family, Objective C, Python, Perl, PHP, ...

- Some languages do both...
  - Java: for down-casting
  - Static type vs. dynamic type

Abstract Data Types (ADT)

- Combining data with associated operations

  Haskell:
  ```haskell
  module Stack (Stack, empty, isEmpty, push, top, pop) where
  empty :: Stack a
  isEmpty :: Stack a -> Bool
  push :: a -> Stack a -> Stack a
  top :: Stack a -> a
  pop :: Stack a -> (a,Stack a)
  ```
Abstract Data Types

- **ML:**
  ```ml
  structure Stack =
  struct
    type 'a stack = 'a list
  val empty = []
  val push = op ::
    fun pop [] = NONE
    | pop (tos::rest) = SOME tos
  end

  Stack.push(100, Stack.empty);
  ```

- **Haskell:**
  ```haskell
  datatype mylist = Nil | Cons of int * mylist
  val l = Cons(10,Nil);

  type occurs on the right-hand side of it's definition.
  ```

Ad-Hoc Polymorphism

- **Polymorphism:** Functions (or operators) can operate on multiple different types.
- **Parametric Polymorphism**
  ```
  a' -> (a' -> b') -> b'
  ```
- **Subtype Polymorphism**
  ```
  S <: T
  ```
- **Ad-Hoc:** not rooted in the type system.
  ```
  E.g., operator overloading
  Arguments to an operator do not need to be related in any form in the type system.
  E.g., “+” for Int addition and String concatenation.
  ```

Recursive Types

- **ML:**
  ```
  datatype mylist = Nil | Cons of int * mylist
  val l = Cons(10,Nil);
  ```
- **Haskell:**
  ```
  data List a = Nil | Cons a (List a)
  ```

Type Classes

- **Expressing operator overloading in the type system**
- **Haskell:**
  ```
  data Tree a = Nil | Leaf a | Branch (Tree a) (Tree a)
  class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool

  instance (Eq a) => Eq (Tree a) where
    Leaf a == Leaf b = a == b
    (Branch l1 r1) == (Branch l2 r2) = (l1==l2) && (r1==r2)
    _ == _ = False
  ```
Existential Types

- Idea: Fixed but not further specified type
  - Think of J.P. Sartre: Existence precedes Essence.
- Method of abstraction

Haskell (extension):
- `data Litem = forall a. (Show a) => Litem a`
- `lst :: [Litem]
lst = [Litem 1, Litem "hello", Litem 'a']`
- `doShow :: [Litem] -> String
doShow [] = ""
doShow ((Litem x):xs) = show x ++ doShow xs`

Warmup: Monoids

- Monoid
  - Set $S$
  - Binary operation $\cdot$
    - Closure: $\forall a, b \in S : a \cdot b \in S$
    - Associativity: $\forall a, b, c \in S : (a \cdot b) \cdot c = a \cdot (b \cdot c)$
    - Identity element: $\exists e \in S : \forall a \in S : e \cdot a = a \cdot e = a$

Category Theory

- Categories = Algebraic Structures
  - Attempt to axiomatically capture commonalities between similar structures
- Category (informal):
  - Collection of Objects
  - Morphisms: Structure-preserving mapping from one structure to another
- Example: Category of Sets:
  - Objects = Sets
  - Morphisms = Functions
CCC

- Axioms are relatively weak
- We often want more restrictive categories with more interesting properties
- Example: Cartesian Closed Categories
  - It has a terminal object
    - \( T \) is a terminal if \( \forall X \in C : \exists \text{ exactly one } f : X \to T \)
  - Any two objects \( X \) and \( Y \) have a product \( X \times Y \) in \( C \)
  - Any two objects \( Y \) and \( Z \) have an exponential \( Z^Y \) in \( C \)

\[ \lambda - Calculus! \]

- Currying is a consequence of the category-theoretical background:
  - \( X \times Y \to Z \quad X \to Z^Y \)

Functors

- In Category Theory: A Functor maps between Categories.
  - Objects to Objects
  - Morphism to Morphism

- In Haskell:
  - The `Functor` class is defined like this:
    ```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```
  - All instances of `Functor` should obey:
    ```haskell```
    ```
    fmap id = id
    fmap (p . q) = (fmap p) . (fmap q)
    ```

Monads

- In Category Theory: Monad = Endofunctor with two natural transformations.
  - Natural transformation: transforming one Functor into another while preserving structure

- If \( C \) is a Category, then a Monad over \( C \) consists of a functor \( T : C \to C \) with natural transformations
  - \( 1_C \to T \)
  - \( T \circ T \to T \)

- In Haskell: A Monad is a
  - Type constructor that defines for every underlying type a monadic type
  - A unit function that maps a value of the underlying type to a value of the monadic type (\texttt{return})
  - A binding operation \( (M t) \to (t \to M u) \to (M u) \) (\textit{represented by >>=})
Monads

- **Identity Monad**
  
  ```haskell
  Id t = t
  return x = x
  x >>= f = f x
  ```

- **Maybe Monad**

```haskell
data Maybe a = Just a | Nothing

return :: a -> Maybe a
return x = Just x

(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
m >>= g = case m of
  Just x -> g x
  Nothing -> Nothing
```

- **State Monad**

```haskell
type State s t = (t, s)

return x = s x

m >>= f = \s -> let (x, s) = m s in f x s
```

- **IO Monad**

```haskell
(>>) :: IO a -> IO b -> IO b

Hello world in Haskell

main :: IO ()
main = putStrLn "Hello World"
```

Uniqueness Types

- **Pure functional languages have their problems**
  - Side effects or “destructive state”
  - function read(File f) returns String

- **Clean: Uniqueness Types as an alternative to Monads**
  - Unique type means at most one reference to it
  - Helps with concurrency
    - (by preventing it...)
  - function readUnique(unique File f) returns (File, String)

Closures

```haskell
function foo(x : int) {
  return x + a;
}
```

- Where does the “a” come from?
  - Context.

- A (lexical) Closure is a combines the function with its referencing environment (free variables).
Closures

- "Capture by reference"
- JavaScript

```javascript
var y = 1;

function foo(x) {
    return x + y;
}

y = 10;

alert(foo(1));
```

Closures

- "Capture by value."
- Java: Anonymous inner classes

```java
void foo() {
    final int x = 42;
    new Thread() {
        public void run() {
            System.out.println(x);
        }
    }.start();
}
```

Continuations

- Traditional functions:
  ```python
def sq(x):
    return x*x
  ```
- Continuation
  ```python
def sq(x,c):
    return c(x*x)
  ```
- Scheme: call/cc
  ```scheme
  (define (sq x)
    (- x 1))
  ```

Overview

- Created in my 2013 PL Honors class